263-0007-00: Advanced Systems Lab
Assignment 1: 100 points
Due Date: Th, March 10th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2022/
Questions: fastcode@lists.inf.ethz.ch

Exercises:

1. (15 pts) Get to know your machine
   Determine and create a table for the following microarchitectural parameters of your computer:

   (a) Processor manufacturer, name, number and microarchitecture (e.g. Haswell, Skylake, etc).
   Solution: Intel(R) Xeon(R) CPU E3-1275 v5 (Skylake).

   (b) CPU base frequency.
   Solution: 3.6 GHz is the nominal CPU frequency.

   (c) CPU maximum frequency. Does your CPU support Turbo Boost or a similar technology?
   Solution: It does support Turbo Boost, and the maximum frequency is 4.0GHz.

   (d) Phase in the Intel's development model: Tick, Tock or Optimization. (if applicable)
   Solution: Tock phase (Skylake).

   Intel's processors offer two assembly instructions to compute scalar floating-point multiplication in single precision, namely FMUL (from x87) and MULSS (from SSE).

   (e) Which instruction is used when you compile code in your computer and why is the other one still supported?
   Solution: MULSS from SSE. FMUL and all instructions from x87 are supported from backward compatibility.

   (f) What is x87 and why is it called that way?
   Solution: It is a floating-point extension of x86. The name originates from the 8087 co-processor used for math operations. Later, Intel integrated it into its instruciton set with the name x87.

   For one core and **without** using SIMD vector instructions determine the following (make sure to use the correct floating-point instruction in (h)-(j)).

   (g) Maximum theoretical floating-point peak performance in flops/cycle.
   Solution: Without SIMD instructions, two FMAs can be issued per cycle. Thus, 4 flops/cycle.

   (h) Latency [cycles], throughput [ops/cycle] and instruction name for both double- and single-precision floating-point subtraction.
   Solution: Latency: 4 cycles. Throughput: 2 per cycle. Instruction: SUBSS(D).

   (i) Latency [cycles], throughput [ops/cycle] and instruction name for both double- and single-precision floating-point division.
   Solution:
   Latency: 11 cycles. Throughput: 0.33 per cycle. Instruction: DIVSS.
   Latency: 13-14 cycles. Throughput: 0.25 per cycle. Instruction: DIVSD.

   (j) Latency [cycles], throughput [ops/cycle] and instruction name for single-precision floating-point approximate reciprocal square root.
   Solution: Latency: 4 cycles. Throughput: 2 per cycle. Instruction: RSQRTSS.

2. (20 pts) Matrix multiplication
   In this exercise, we provide a C source file for multiplying an $n \times n$ matrix with its transpose and a C header file to time the matrix multiplication using different methods under Windows and Linux (for x86 compatible systems). Inspect and understand the code and do the following:

(a) Using your computer, compile and run the code. Compile with the highest level of optimization provided by your compiler (with GCC, compile with the flag `-O3`). A modern compiler will automatically vectorize this very simple routine. Ensure you get consistent timings between timers and for at least two consecutive executions. Don't forget to disable Turbo Boost. (No need to answer anything here)

(b) Inspect the `compute()` function in `mmm.c` and answer the following:

i. Determine the exact number of floating-point additions and multiplications that it performs.
Solution: The code performs $2n^3$ floating-point operations.

ii. Determine an upper bound on its operational intensity (consider only reads and assume empty caches).
Solution:
$W(n) = 2n^3$ and $Q(n) \geq 2 \cdot 8n^2$. Thus, $I(n) \leq n/8$ flops/bytes.

(c) For all square matrices of sizes n between 100 and 1500, in increments of 100, create a performance plot with n on the x-axis and performance (flops/cycle) on the y-axis. Create three series such that:

i. The first series has all optimizations disabled: use flag `-O0`.

ii. The second series has the major optimizations except for vectorization: use flags `-O3` and `-fno-tree-vectorize`.

iii. The third series has all major optimizations enabled: use flags `-O3`, `-ffast-math` and `-march=native`.

Solution:

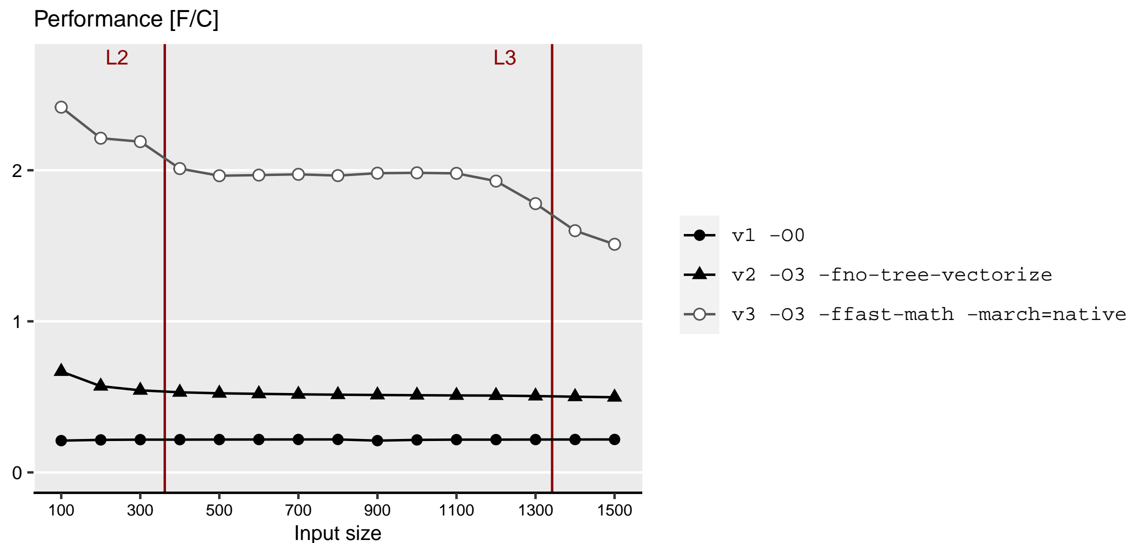

Figure 1: Plots resulting from execution of `mmm.c` (vector peak performance: 16 f/c for the given flags).

(d) Discuss performance variations of your plots and report the highest performance that you achieved.

Solution:

i. Non-optimized (`v1`): This results in machine code that is neither optimized or vectorized. This is nice for debugging. However, the performance is low and flat across problem sizes.

ii. Optimized but non-vectorized (`v2`): The performance is better than in the previous case. However, the performance suffers due to the limited amount of ILP caused by inter loop dependencies.

iii. Fully optimized (`v3`): The `-ffast-math` flag enables ILP which is combined with vectorization and significantly improves performance. The computation performs well for small problem sizes but performance suffers greatly as soon as the matrices no longer fit in the L3 cache.

3. (25 pts) Performance analysis and bounds

   Assume that vectors u, x, y and z of length n are implemented using double precision floating-point and combined as follows:

$$z_i = u_i \cdot u_i - x_i \cdot y_i \cdot (u_i + x_i)$$

   We consider a Core i7 CPU with a Skylake microarchitecture. As seen in the lecture, it offers FMA instructions (as part of AVX2). Recall that we consider cost of the FMA instruction as two floating-point operations (an addition and a multiplication). Assume the bandwidths that are given in the additional material from the lecture: Abstracted Microarchitecture. Assume that no optimization is performed that simplifies floating-point arithmetic (i.e. `-ffast-math` flag is not used). Answer the following and justify your answers.

   (a) Define a suitable detailed floating-point cost measure $C(n)$.

   Solution:

$$C(n) = C_{add} \cdot N_{add} + C_{mult} \cdot N_{mult}.$$

   It is also fine to separate mults and adds.

   (b) Compute the cost $C(n)$ of the computation.

   Solution:

$$N_{add} = 2n,$$
$$N_{mul} = 3n,$$
$$C(n) = C_{add} \cdot (2n) + C_{mul} \cdot (3n).$$

   (c) Consider only one core without using vector instructions (i.e. using flag `-fno-tree-vectorize`) and determine a hard lower bound (not asymptotic) on the runtime (measured in cycles), based on:

   i. The throughput of the floating-point operations. Assume that no FMA instructions are used. Be aware that the lower bound is also affected by the available ports offered for the computation (see lecture slides).

   ii. The throughput of the floating-point operations where FMAs are used to fuse an addition and a multiplication (i.e. `-mfma` flag is enabled).

   iii. The throughput of data reads, for the following two cases: All floating-point data is L2-resident, and all floating-point data is RAM-resident. Consider the best case scenario (peak bandwidth and ignore latency). Note that arrays that are only written are also read and this read should be included.

   Solution: We can obtain bounds by examining which execution ports the instructions are scheduled to and the throughputs of those instructions.

   i. The instruction mix in this case consists of $2n$ floating-point additions and $3n$ floating-point multiplications. Both, mults and adds can be scheduled in Ports 0 and 1. Thus, a lower bound on the runtime is $2.5n$ cycles.

   ii. We can only fuse the subtraction into an FMA. Thus, we have n FMA instructions, n additions and $2n$ multiplications. FMAs can also be scheduled in either Port 0 or Port 1. Thus, resulting in a lower bound of $2n$ cycles.

   iii. Abstracted Microarchitecture shows peak bandwidth of L2, and an estimate for the RAM throughput. In the computation, at least $4n$ doubles have to be read in total. Thus, $r_{L2} \geq \frac{4n}{8} = \frac{n}{2}$ and $r_{RAM} \geq \frac{4n}{2} = 2n$.

   (d) Determine an upper bound on the operational intensity. Assume empty caches and consider only reads but note: arrays that are only written are also read and this read should be included.

   Solution: The operational intensity is $I(N) \leq \frac{5n flops}{8(4n) bytes} = \frac{5}{32}$ flops/byte.

4. (25 pts) Basic optimization

Consider the following function:

```
1 void comp(double *x, double *y, int n) {
2     double s = 0.0;
3     for (int i = 0; i < n; i++) {
4         s += x[i] * x[i] + y[i];
5     }
6     x[0] = s;
7 }
```

(a) Create a benchmarking infrastructure based on the timing function that produces the most consistent results in Exercise 2 and for all two-power sizes $n = 2^4, \ldots, 2^{23}$ create a performance plot for the function `comp` with n on the x-axis (choose logarithmic scale) and performance (in flops/cycle) on the y-axis. Randomly initialize all arrays. For all n repeat your measurements 30 times reporting the median in your plot. Compile your code with flags `-O3 -fno-tree-vectorize`.

(b) Considering the latency and throughput information of floating-point operations in your machine, and the dependencies in `comp`, derive an upper bound on the performance (flops/cycles) of `comp` when using the specified flags in (a), i.e., when FMA and vector instructions are disabled (flag `-mfma` is not used and `-fno-tree-vectorize` is enabled).

Solution:

The runtime is limited by an inter loop dependency when accumulating the values in `s`. The latency of addition is 4 cycles (Skylake). Thus, $T(n) \geq 4n$. Since $W(n) = 3n$, the performance is upper bounded by $\pi(n) \leq 0.75$ flops/cycle.

(c) Perform optimizations that increase the ILP of function `comp` to improve its runtime. It is not allowed to use FMA or vector instructions. Add the performance to the previous plot (so one plot with two series in total for (a) and (c)). Compile your code with flags `-O3 -fno-tree-vectorize`.

(d) Discuss performance variations of your plot and report the highest performance that you achieved.

(e) Enroll and submit the code of your optimized function in Code Expert. Carefully read and follow the instructions given in Code Expert to submit your code.

Solution:

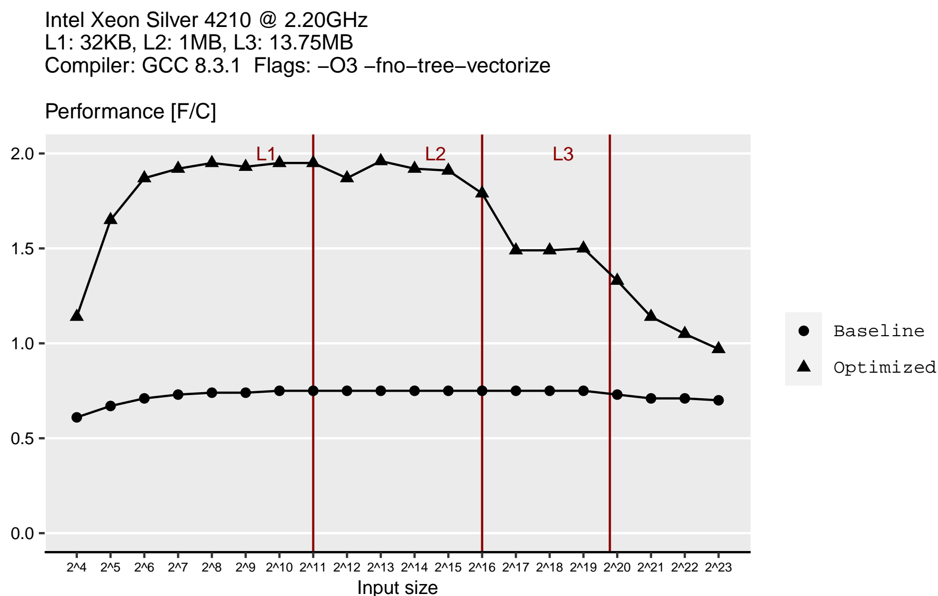

Figure 2: Performance plot (peak performance: 2 f/c for the given flags).

In the original code, the performance suffers from inter loop dependency which limits the amount of ILP. Thus, the performance is 0.75 flops/cycle across all problem sizes and it's consistent with the upper bound derived in (b). Unrolling the loop and using separate accumulators increases the ILP. For this case, we see that performance varies across problem sizes. Performance is great when the data fits in cache, and becomes worse as the size of the data grows. We can even see "steps": performance is greatest when the data fits in L1, and becomes incrementally worse as it no longer fits in subsequent levels of cache. The maximum performance achieved is 1.97 flops/cycle.

5. (10 pts) ILP analysis

   Consider the following computation:

```
1 double artcomp(double a, double b, double c, double d) {
2     double r;
3     r = ((a * b + c) * b + c) - a / d;
4     return r;
5 }
```

   Make the same assumptions as in Exercise 3, i.e., consider a Skylake processor, only one core without using vector instructions (using flag `-fno-tree-vectorize`), and assume that no optimization is performed that simplifies floating-point arithmetic (i.e. `-ffast-math` flag is not used). Thus, it is not allowed to apply associativity and distributivity laws to rearrange the computation. Determine hard lower bounds (not asymptotic) on the runtime (measured in cycles), based on the following. Note that it may be useful to draw the dependency graph of the computation. Justify your answers.

   (a) The latency, throughput and dependencies of the floating-point arithmetic operations. Assume that no FMA instruction is generated (i.e. `-mfma` flag is not used). Be aware that the lower bound is also affected by the available ports offered for the computation (see lecture slides).

   Solution: All operations are in the critical path except the division (see Figure 3). Thus, the runtime is at least **20 cycles**.

   (b) The latency, throughput and dependencies of the floating-point arithmetic operations when FMAs are enabled to fuse an addition and a multiplication (i.e. `-mfma` flag is used).

   Solution: For this case, the division and the final subtraction are in the critical path (see Figure 3). Thus, the runtime is at least **18 cycles**. For some inputs, the division has a latency of 13 cycles (see Agner's instructions table). So we also consider as correct a lower bound of 17 cycles.

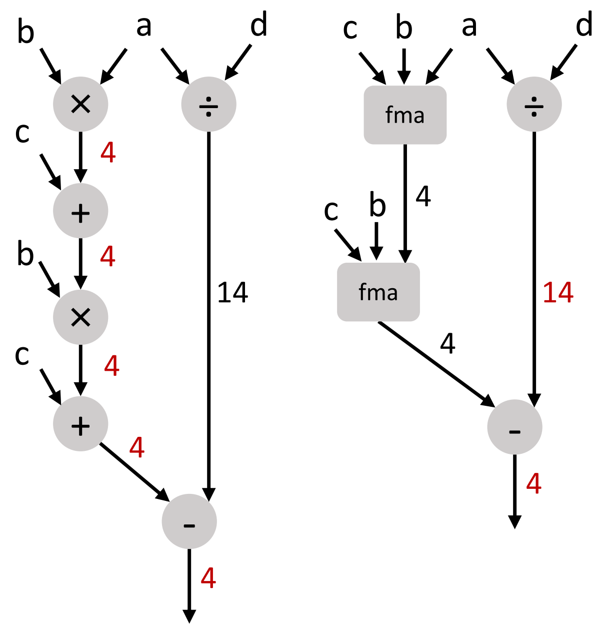

Figure 3: Dependency graph for `artcomp`.